# System programming (03KTOOT)

## Laboratory #3: **Threads**

Laboratories are mandatory. Each student must submit the work (exclusively in .zip format, with text in .pdf and code (if required) in a format able to be run on the LABINF computers) following the rules published on the web portal. Please remember that for each laboratory the final mark can get from -2.0 (not submitted or with severe flaws) up to +0.5 points (excellent work). If all 4 laboratories are not submitted 8 points are lost.

**The deadline for the third laboratory is Dec 18th 2015 at 7 p.m**

**Exercise 1:** Write a C program that creates a large number of processes. Repeat the operation with threads. Find the maximum number of processes and threads that it is possible to create under the chosen architecture and operating system. Compute the time needed to create and destroy processes and threads. Is it better to reboot the computer after such experiments?

**Exercise 2:** Write a simple counting program, with two threads. The first thread should loop, incrementing a counter as fast as it can, while the second thread should occasionally read the counter value and print its value. Are there concurrency problems?

**Exercise 3:** Write a program who creates two threads; the first will read a file whose name is given as a **command line parameter** and will send the data to another thread who will write data on a second file (whose name is also given as a command line parameter). What is the difference with the similar process-based exercise of Lab #2?

**Exercise 4:** Extend exercise 2 in order to protect the counter variable with a mutex; is it possible to use a mutex to protect an integer variable used as a counter semaphore? Which are the limitations?

**Exercise 5:** Write a program that creates two children for a total of 3 processes. Each of the children should now create its own threads. Are there any limitations in data exchanging? Try to do an exec from one of the children; what happens to threads?

**Exercise 6:** Write a program with 4 threads and 1 counter. One thread should wait for input (e.g., a scanf instruction); as soon as an input value is available, the value should be added to the counter. The other threads should stay in a loop, in which they should:

1- wait for the counter to be greater than zero
2- decrement the counter and print a message with their own ID and the counter value
3- sleep for one second (see Linux Program Development book).

**Exercise 7:** Write a multi-thread program that creates 10 threads. The threads are numbered from 0 to 9 and receive their order number from the main program. Each thread generates a random number in the range 1-10 and stores it in a shared array in the position indicated by its order number.

When all threads complete their execution the main program computes the sum of all generated numbers and prints it on the screen.

**Exercise 8**: Write a multi-thread program that evaluates the following math series:

$$\sum_{0}^{N} x^i$$

The main program receives the x and N values as input (try not to set a max value for N) and creates the set of necessary threads. Each thread evaluates a single $x^i$ instance, referring to its index of creation (e.g., thread 1 evaluates $x^0$, thread 2 evaluates $x^1$, and so on) and adds it to the final results. Once all the threads complete their job, the main program displays the final result. Is there any difference with respect to the similar problems solved in Lab #2?

**Exercise 9**: Write a multi-thread program to search if a number N is a prime number by checking if it is divisible by any number in the range < 2, N/2 >. The program receives in input the number N and the number P of threads to use.
The program must create P threads and assign to each of them a subset of possible divisors to test. Each thread checks if N is divisible by any of the assigned divisors.
The main program continuously checks the result of the generated threads. As soon as a thread detects that the number is not prime, the main program must stop the executions of all generated threads and exit.
Once the program it's ready, try its execution with big numbers and different values of P to understand how the execution time changes. Printing the start and end time is a good idea to trace the execution time